Homework 3

COMP221 Spring 2024 - Suhas Arehalli

Complete the problems below. This assignment is out of 100 points, and the point values of each question. Note that point values are roughly inversely correlated with expected difficulty.

- You may complete the implementation section using your choice of Python, C, or Java.
- Write up your solutions to all parts of this homework using some typesetting software (LaTeX/Overleaf is recommended, but any typeset PDF is acceptable).
- You'll submit this assignment through Github Classroom (link on Moodle), with your written solutions in a pdf named "[LASTNAME1]_[LASTNAME2]_Homework1.pdf".
- **The due date for this assignment will be posted on the course website.**
- **You may submit assignments in pairs.** Put both peoples' names at the top of your submission.
  - **Both people are expected to have contributed to all parts of the assignment** – work together, and make sure you and your partner are both convinced of your solutions.
  - If you discussed problems with other students, list their names at the top of your assignment (there will be no penalty for this — it's encouraged!).
  - However, don't look at other group's written solutions/code, or share your written solutions/code with other groups. **In addition, do not search around the internet for solutions to these problems.** Seek help from your partner and course staff, the textbook, and course materials. Consult the syllabus for more details on academic integrity policies.

Note that we may not have covered all of the topics involved in this assignment at the point of release. This is to give you time to complete the parts you can as early as you can. I recommend you make note of parts we haven't covered yet so you can come back to them.

1 Implementation (49pts)

1. Let's start by building some prerequisite data structures...
   (a) Implement a (binary) heap-based priority queue, using an array as the underlying tree data structure. Remember that a complete tree can be represented in a space-efficient way by having the root at index 0, it's children at indices 1 & 2, the children of those nodes at indices 3–6, and so on. I.e., the children of the node at index i lie at indices $2i + 1$ and $2i + 2$ (you can prove that this is true!). Make the initial size of the array such that no

errors will be encountered for < 1000 elements. Your priority queue should implement an enqueue and dequeue operation.

   (b) Implement an AdjacencyList representation of a weighted, *undirected* graph. You need only implement the operations you need for the following algorithms, and may adapt implementations you completed in COMP128 (though I suggest you revisit those carefully).

2. And now let's implement some graph algorithms...
   (a) Use the sample file processing code to construct a graph containing the edges in a provided file (two samples are provided, though you should create more using this same format). Implement Dijkstra's algorithm to find and print to command-line the *cost* of the shortest path between the start and end vertices provided by the command-line input when the ALG argument is "SP." *(15pts)*
   (b) Implement Prim's algorithm **as we saw in class OR using some optimizations from the textbook**, such that when the ALG argument is "MST-Lin," your code finds and prints to the command-line the *cost* of the minimum spanning tree of the provided graph. *(20pts)*
   (c) Implement Prim's algorithm **using a priority queue**, such that when the ALG argument is "MST-PQ," your code finds and prints to the command-line the *cost* of the minimum spanning tree of the provided graph. *(14pts)*

2 Algorithmic Analysis & Design (50pts)

1. **Shortest Paths with Negative Weights** *(25pts)*

   In class, we saw Dijkstra's algorithm, which found shortest paths in greedy fashion. Unfortunately, Dijkstra's can fail when a graph has negative edge weights. When we looked at the All-Pair Shortest Path problem, we saw that Floyd-Warshall can find shortest paths as long as there are no negative weight *cycles*. In this problem, we'll adapt the *relaxation*-based approach used by Floyd-Warshall to solve the single-source shortest path problem in graphs with negative edge weights!

   Like in Floyd-Warshall, if t is not reachable from s, we will adopt the convention that the length of the shortest path between s and t is notated as ∞. ∞ should interact with mathematical operations as you might expect: It is greater than all real numbers, and anything added to ∞ will result in ∞.

   (a) Recall from COMP128 that a path is *simple* if it does not repeat vertices. We define paths as sequences of edges $(v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k) \subseteq E$, so we can say a path is simple if no vertex v_i is repeated $(\forall 1 \leq i, j \leq k, v_i \neq v_j)$.

   Prove that for any directed graph $G = (V, E)$, if a shortest path exists from some $s \in V$ to $t \in V$, that shortest path is *simple. (4pts)*

   ****Hint****: A path P from s to t is *not* a shortest path if there exists a shorter path P' from s to t. If you can show that for *any* path P from s to t, there exists a shorter path P' from s to t, then you've proven that there is not shortest path from s to t.

   (b) Using your result from (a), what is the maximum possible *number of edges* in the shortest path from $s \in V$ to $t \in V$. *(1pt)*

(c) Let $\delta(s,t)$ for $s,t \in V$ be an *upper bound* on the length of the shortest path from s to t, and let $SP(s,t)$ be *actual* length of the shortest path from s to t. Observe that this means that $\forall s,t \in V$, $SP(s,t) \leq \delta(s,t)$.

Prove that if $(v,t) \in E$, $SP(s,t) \leq \delta(s,v) + c((v,t))$. *(3pts)*

(d) Consider the following pseudocode.

```
Let $P$ be the shortest path from $s$ to $t$.
for $t \in V \setminus \{s\}$ do
    $\delta(s,t) \leftarrow \infty$
end for
$\delta(s,s) \leftarrow 0$
for $(v,w) \in P$ do
    $\delta(s,w) \leftarrow \min(\delta(s,v) + c((v,w)), \delta(s,w))$
end for
```

Prove that after this code is run, $\delta(s,t) = SP(s,t)$. *(4pts)*

(e) Of course, that code isn't particularly impressive, since we assume we already know P! Consider this variation on that pseudocode that doesn't make that assumption:

```
for $t \in V \setminus \{s\}$ do
    $\delta(s,t) \leftarrow \infty$
end for
$\delta(s,s) \leftarrow 0$
for $i \leftarrow 1...k$ do
    for $(v,w) \in E$ do
        $\delta(s,w) \leftarrow \min(\delta(s,v) + c((v,w)), \delta(s,w))$
    end for
end for
```

Note that k is not defined here! Choose a value of k and then prove (based on that value of k) that if there exists a shortest path from s to t, we end with $\delta(s,t) = SP(s,t)$. *(5pts)*

****Hint**:** Consider the first iteration of the outer loop: What happens when (v,w) is the first edge of the shortest path? What happens when it isn't? What does that tell you about what happens after the 1st iteration? The ith iteration?

(f) Suppose that we find that *after* that snippet of code is run, that for some $(v,w) \in E$, $\delta(s,w) > \delta(s,v) + c((v,w))$. Prove that this means G contains a negative weight cycle reachable from s. *(4pts)*

(g) Provide full pseudocode for a shortest path algorithm SHORTESTPATH using the above snippets. It should take as an argument a graph $G = (V,E)$, a cost function $c : E \to \mathbb{R}$, and source and target vertices $s,t \in V$ and return the distance of the shortest path from s to t if there are no negative cycles reachable from s and $NULL$ otherwise. *(4pts)*

2. **Arbitrage** *(12pts)*

In finance, arbitrage refers to a strategy where one finds a sequence of transactions one can make instantaneously that results in a profit. In this problem we'll be looking at efficient algorithms to determine if arbitrage is possible when exchanging currencies.

Suppose you have an n x n matrix R that, for any two currencies $c_1, c_2 \in C$, gives you an exchange rate $R_{i,j}$ such that 1 unit of currency c_1 can be exchanged for $R_{i,j}$ units of currency

c_2. R is said to be *arbitrage-free* if there is no sequence of exchanges such that you can make a profit (end up with more of an initial currency than you started with).

For example, suppose $R_{1,2} = 0.5$ and $R_{2,1} = 2.1$. If you started with 1 unit of currency 1 ($1c_1$), you could exchange that for $0.5c_2$, and exchange that back for $(2.1)(0.5)c_1 = 1.05c_2$! By making this sequence of exchanges, you have profited $0.05c_1$, and thus in this case, R is *not* arbitrage-free.

(a) Reframe this as a graph problem. State what edges and vertices correspond to. What does it mean (in graph terms) for R to be arbitrage free? *(5pts)*

(b) Provide pseudocode for an efficient algorithm for determining if R is arbitrage free. You may use graph algorithms we've discussed in class without reproducing their pseudocode. *(4pts)*

****Hint****: Consider that $-\log k$ is negative iff $k > 1$, and that $\log(k_1 \cdot k_2) = \log k_1 + \log k_2$.

(c) Suppose R is not arbitrage-free. As a trader, you want to find the sequence of currency exchanges that maximizes your profit. Assume that every exchange can only be conducted once (otherwise, one could repeat a profitable sequence endlessly!). Provide a modification to your pseudocode in the previous question to compute the percent-profit gained from conducting that sequence of trades. *(3pts)*

For instance, in the example given above, you gain 5% profit, making $0.05c_1$ for every $1c_1$ you trade.

3. **Making Change on a Global Scale** *(13pts)*

On the topic of finance, let's revisit a problem from the last homework: Suppose we have a set of coin denominations C with which we're trying to make change totaling to k minimizing the number of individual coins we use.

However, unlike the problem in the previous homework, we want to generalize our solution to *any* possible coin system, not just the one in the United States!

(a) Prove that a greedy algorithm that selects the largest coin possible will not work for all sets of coin denominations C. That is, construct C such that the greedy approach fails to find the minimal choice of coins. *(2pts)*

(b) Provide pseudocode for a dynamic programming algorithm that computes the minimum number of coins needed to make k units of change that works for *any* coin system. *(4pts)*

(c) Determine the time *and space* complexity of the algorithm you provided. *(3pts)*

(d) Prove that this algorithm is correct. *(4pts)*